ERB-966

NRCC No. 23030

National Research Council Canada
Conseil national de recherches Canada

Division of Electrical Engineering
Division de génie électrique

USING THE HARMONY OPERATING SYSTEM

W.M. GENTLEMAN

DECEMBER 1983

Canada

Copyright 1983 by
National Research Council of Canada

Permission is granted to quote short excerpts and to reproduce figures and tables from this report, provided that the source of such material is fully acknowledged.

Copyright 1983 par
Conseil national de recherches du Canada

Il est permis de citer de courts extraits et de reproduire des figures ou tableaux du présent rapport, à condition d'en identifier clairement la source.

Additional copies are available free of charge from:

Editorial Office, Room 301
Division of Electrical Engineering
National Research Council of Canada
Ottawa, Ontario, Canada
K1A 0R8

Des exemplaires supplémentaires gratuits peuvent être obtenus à l'adresse suivante:

Bureau des publications, Pièce 301
Division de génie électrique
Conseil national de recherches du Canada
Ottawa (Ontario) Canada
K1A 0R8

USING THE HARMONY OPERATING SYSTEM

W.M. Gentleman

ERB-966
NRCC No. 23030

DECEMBER 1983

TABLE OF CONTENTS

	Page
ABSTRACT	v
RÉSUMÉ	vii
WHAT IS HARMONY? WHAT IS IT NOT?	1
WHAT IS REALTIME?	2
TASKS	4
COMMUNICATIONS AND SYNCHRONIZATION	7
STORAGE CLASSES	10
STREAM INPUT/OUTPUT	11
OTHER INPUT/OUTPUT, INTERRUPTS, and SERVERS	14
USER'S VIEW OF A PROGRAM	17
SUPPORT TOOLS	19
Listing	20
Examine	20
Bound	20
Debug	20
PERFORMANCE	21
REFERENCES	22
APPENDIX A. A SAMPLE MULTITASK PROGRAM	23
APPENDIX B. A SAMPLE MULTITASK MULTIPROCESSOR PROGRAM	26
APPENDIX C. DECLARATIONS	30
APPENDIX D. SUPPLIED TASKS	32

ABSTRACT

Harmony is a multitasking, multiprocessing operating system for realtime control. It can also be used to implement "smart peripherals" to be used with itself or with some other operating system, and it provides a simple and inexpensive vehicle for experimenting with parallel computation. It is a portable system in that with moderate effort it can be realized on any computer of appropriate architecture. The current realization runs on the Chorus multiprocessor, a Motorola 68000 based Multibus machine. The operating system is mostly written in C, and user programs must currently also be written in C. Programs written to run in Harmony, however, look very different from C programs written to run in Unix or some other conventional operating system.

This document is a user's manual for writing programs to run in Harmony. It not only describes the facilities available and the syntax for using them, but also outlines something of the philosophy of why these facilities are provided and how they are intended to be used. Moreover, some of the non-programming aspects of the system, such as the mechanics of running a program, are discussed.

RÉSUMÉ

Harmony est un système d'exploitation multitâche et multitraitement pour commande en temps réel. Il peut aussi servir à la mise en application de "périphériques intelligents" qu'il commandera ou qui seront commandés par un autre système d'exploitation, et il permet d'effectuer de façon simple et à peu de frais des expériences de calcul en parallèle. Il peut être considéré comme un système portatif, puisque moyennant un effort modéré, il peut être appliqué à un ordinateur de type approprié. La version actuelle est utilisable sur le multiprocesseur Chorus, une machine Multibus basée sur le Motorola 68000. La majeure partie du système d'exploitation est écrite en langage C et, actuellement, les programmes d'utilisateurs doivent aussi être écrits en langage C. Les programmes écrits pour être traités par le Harmony ont cependant une apparence très différente de celle des programmes écrits en langage C devant être traités par le Unix ou un autre système d'exploitation ordinaire.

Le présent document est un guide de l'utilisateur portant sur l'écriture de programmes qui seront traités par le Harmony. Il ne décrit pas seulement les moyens disponibles et les règles de syntaxe permettant de les utiliser; il donne également les grandes lignes de la raison d'être de ces moyens et indique dans quelle optique on a conçu leur utilisation. De plus, il traite de certains aspects du système non reliés à la programmation, par exemple les techniques d'exécution d'un programme.

USING THE HARMONY OPERATING SYSTEM

W. Morven Gentleman

WHAT IS HARMONY? WHAT IS IT NOT?

Harmony is a multitasking, multiprocessing operating system for real-time control. (Each of these terms will be examined below.) It is a portable system, in that with moderate effort it can be realized on any computer of appropriate architecture. The current realization runs on the Chorus multiprocessor, a Motorola 68000 based Multibus multiprocessor using Omnibyte OB68K1A single board computers. The operating system is written in C (plus a small amount of assembler language) and user programs must currently also be written in C, although the system is designed to support eventually other languages such as Fortran and Pascal. Care has been taken to avoid dialect or library dependence on any specific C compiler - Whitesmiths' C compiler is what is used currently.

Although C is used as the programming language, programs written to run in Harmony look, and are thought of, very differently from conventional C programs which are written to run in Unix or some other conventional operating system. This difference goes beyond the question of what functions are available in the standard libraries and will be discussed at length below.

Harmony is an open system. This is a description that can be used at many levels. In the case of Harmony, it is intended to imply that it is easy to use the system on many different configurations of hardware, and in particular to support peripherals not thought of when Harmony itself was originally designed. It is straightforward to integrate a new device into both the physical level of connection and the higher levels of software abstraction.

One major aspect is that Harmony is not designed to support a program development environment, and in particular it is not a multiuser timesharing system. Harmony devotes the entire resources of the machine to running a single multitask applications program, and is bound in with that program at link time. Program development is done on some host computer, running its own operating system, and test executions are performed by downloading executable images into target processors. (Production programs instead might well be in ROM and be started by power up or reset.) In particular, Harmony programs cannot assume they have been started by a command interpreter like the Unix shell, and that they have command line arguments and redirected I/O. Indeed, they may not have terminal I/O at all. Some of the specialized tools used in the program development environment to assist in building Harmony programs will be described later.

WHAT IS REALTIME?

Returning to examine the terms used in the initial sentence, consider first the term "realtime". Realtime programming has two independent aspects. First, programs must interact with the real world, and second, time itself may be a critical resource.

What makes interacting with the real world different from other kinds of programming is primarily that relevant events in the real world can happen independently of, and asynchronously with, the state of computation of the program. Conventional programming assumes a master/slave relationship between the computer and the rest of the world - things only happen when requested by the computer and are then guaranteed to happen, typically synchronously with the computation. This relationship does not hold for programs that interact with the real world. Events in the real world are often spontaneous, as seen by a program, and anticipated events often do not happen. There are often relationships between events that the program cannot be aware of. Indeed, the situations that can arise in the real world are often so varied that it is unrealistic to attempt to anticipate them all, and thus programs must cope with unexpected events.

Unless we are willing to limit programs to tracking a single sequence of events and to busywait for each event in the sequence to happen, we need a mechanism for expressing asynchronism in the program, so that the asynchronism in the program can model the asynchronism in the real world and be synchronized with it. This mechanism must be understandable, well defined, and efficient, so programs can be designed using it and can be reasoned about so their behaviour will be known. The mechanism that Harmony provides is tasks (also called processes in some other systems). It will be described below.

A secondary aspect of interacting with the real world is that exotic peripherals are often required to sense or affect things of interest. Not only do these exotic peripherals require special command interfaces, because they do not fit the traditional I/O model, but they often require complex algorithms in order to be used, because they are flakey: sensors do not reliably return correct values, controls do not reliably accomplish the desired effect. Error recovery is thus also intrinsic to interacting with the real world. Both error recovery and coping with unanticipated events are considerably simplified if resources can be allocated and deallocated dynamically. Such resources include tasks themselves, where instantiations must be created and destroyed dynamically, as well as blocks of memory of various sizes which are allocated to and freed by tasks on demand, and connections between tasks and peripherals, which are opened and closed on demand.

Working with time itself as a critical resource is different from other programming in several ways. In conventional programming, time is irrelevant except perhaps in terms of overall efficiency considerations, or for recording and metering. Time can become a critical resource if, for instance, an external event requires a control signal in response within a guaranteed time, or if an external event produces volatile data which must be accepted soon enough so they are not lost, or if periodic activity must be performed at regular intervals with little jitter, or if performance requirements indicate need for explicit parallelism between computation and external activities or between different computations, or if parallelism is intrinsic to the computation, as when stepping motors in orthogonal directions must operate simultaneously to produce a diagonal movement. Of course time is only a critical resource when the constraints are sufficiently stringent, given the available computational power, that conformance must be planned for and will not happen by accident.

Sometimes, when time is a critical resource, time is of the essence and actually defines or redefines the computation. For example, in the case above involving periodic activity, if new sensor data are not available in time for the next cycle of computation, previous data must be extrapolated. Situations dealing with motion, and predicting trajectories, are often like this. More commonly, when time is a critical resource, the requirements translate into scheduling control. There is the choice of which computation to do first, when several are possible. When some computation becomes possible, there is the question of which current computations should be run to completion and which computations should be preempted. There can also be the necessity of ensuring the proper sequencing of otherwise independent computations. In Harmony, these things are achieved by dividing the program into tasks and using the rules for communication and synchronization between tasks, and the rules, including priority, which govern the scheduling of tasks on each processor. (These rules will be detailed below.)

One common characteristic when time is a critical resource is that no unanticipated delays in performing a computation are acceptable, that time performance must be guaranteed. This implies that no algorithms with unbounded execution time can be used, and apparently conflicts with the requirement from interacting with the real world that resources be dynamically allocated. The resolution of this dilemma is to preallocate resources that will be required at critical times, doing the dynamic allocation at times which are not critical. For example, execution stacks are preallocated out of contiguous memory, rather than having stack frames allocated on demand from a heap, so the cost of calling a function is bounded.

TASKS

At several points in the foregoing discussion, the concept of tasks has been alluded to, and it is now appropriate to define it. Tasks are a concept not used in conventional programming, not even in ordinary C programs in Unix, but they are the heart of programming in Harmony. A program in Harmony is made up of one or more, but typically many, tasks. A task is best thought of as like a subroutine, except that an instantiation of it must be explicitly created, and once it has been created, it executes independently of, and in parallel with, the task that created it. (Actually, tasks that are completely independent of each other are not of very much use, so a mechanism is provided for tasks to communicate and synchronize with each other. That mechanism is message passing, which will be described shortly.) Each task itself executes sequentially and synchronously, like a conventional program - the asynchronism and parallelism is strictly between tasks.

Executing in parallel is, of course, only strictly possible when the computer is a multiprocessor, and the tasks are executed by different processors. There are two classical reasons for using multiprocessors: to increase total computing capacity, and to have redundancy to provide reliability in case of hardware failure. In realtime systems, there is a third reason: to guarantee responsiveness, by ensuring that a processor will be available to service an event soon enough. Harmony thus, as stated at the beginning, supports multiprocessing. When a task is created, the processor on which it will execute is specified, and this need not be the same as the processor on which the task requesting the creation is executing. A task competes for scheduling only with other tasks that run on the same processor. Note that the Harmony model of tasks tied to specific processors is different from the model used in some other systems, where any processor can execute any task. The Harmony model recognizes that processors may have local resources, such as memory or peripheral IC's, that can only be accessed by tasks executing on that processor, or that are much more efficient if so accessed. It also facilitates different processors executing different instruction sets. The hardware assumptions Harmony makes are discussed elsewhere.

Since multiple tasks can be created to run on any given processor, Harmony must provide multitasking (also called multiprogramming) to share the processor amongst these tasks. Timesharing systems typically do this with timeslicing and round-robin scheduling, but this would be inappropriate for realtime, given the requirements for scheduling control mentioned above. Instead, each processor supports a separate priority scheduling system. Each task has a fixed priority level. There is a separate FIFO ready queue for each priority level, and all tasks of that priority level which are not blocked are in that queue (blocking actions will be

described later). The first task on the highest priority nonempty queue is the active task, i.e., the one which is executing. It continues to execute until it blocks or is preempted by a higher priority task becoming ready. That is, Harmony tasks run "natural break" and support preemption. Priority, as used in realtime systems, does not refer to "importance", but rather to "urgency". The ideal effect of priority is that the scheduling behaviour of a task can be understood by understanding only those tasks at the same or higher priority levels - the existence of lower priority tasks, much less their behaviour, can be ignored. In particular, if there is only one task at the highest priority, strict bounds can be placed on scheduling delays and jitter.

A task is thus the unit of sequential and synchronous execution. A task is also the unit of resource ownership. All resources, such as memory blocks, peripherals, or I/O connections, are owned and managed by some task or other.

For a task to be created, there must be a corresponding task template, which specifies the essential parameters of the task. (Multiple instantiations of the task defined by a given template can exist at the same time.) A task template is the following struct:

```
struct TASK_TEMPLATE
  {
    unsigned    GLOBAL_INDEX;
    void        (*ROOT)();
    unsigned    STACKSIZE;
    unsigned    PRIORITY;
    unsigned    LOCAL_TASK_MANAGER;
  }
```

The fields in this struct are as follows:

GLOBAL_INDEX is an integer identifying this template. It must be unique over all templates on all processors for this program. In the current configuration it must be between 1 and 235. It is wise to use only larger values, say over 50, and also to use symbolic constants instead of literal integers. The values 1, 2, and 3 are reserved: 1 is the first user task (i.e., it calls procedure main()), 2 is the directory task (i.e., it provides task id's corresponding to symbolic names), and 3 is the gossip task (i.e., it provides a general reporting and logging mechanism for other tasks). The system itself creates on processor 0 one instance of each of these three tasks.

ROOT — is the function that is the code which the task will execute. (This function can, of course, call others.) It is a normal parameterless function with no return value. If the function ever returns, the task is destroyed but the parent is in no way notified. More than one task template might have the same root function.

STACKSIZE — Each task has its own stack, which is preallocated when the task is created. Unfortunately, many compilers, including Whitesmiths', do not generate code to check for stack overflow. A program development tool, the stack bounder, helps with the choice of this value.

PRIORITY — Task priority is specified by an integer; larger is lower priority. Values 0 through 4 correspond to hardware interrupt levels 5 through 1 on the Motorola 68000. The number of priority levels is not fixed, but having too many levels gains little and adds overhead.

LOCAL_TASK_MANAGER — is a field stuffed by the system. It identifies the local task manager that creates and destroys instances of this task. Since each processor has its own local task manager, this effectively determines which processor executes these instances.

The executable image for each processor is compiled and linked separately in Harmony. For each processor, that image is defined, in part, by the declaration of a vector of task templates, _Template_list, that determines what tasks can be created on this processor. The vector is terminated by a null task template, i.e., one in which all fields are zero.

Tasks are created by calls to the _Create primitive, specifying a template index.

```
id = _Create( global_index );
```

The _Create primitive returns an unsigned id, which is used to identify this particular instantiation in future, e.g., for message passing, task destruction, etc.

Two functions are provided to obtain other id's that are often wanted: the id of the task itself and the id of the task which created this one.

```
id = _My_id();
id = _Father_id();
```

As described above, a task is destroyed if its root function returns. It can also destroy itself by calling the _Suicide primitive.

```
Suicide();
```

More generally, any task can be destroyed by calling the _Destroy primitive.

```
_Destroy( id );
```

Destruction implies that the task is stopped, its id made invalid, and that its resources are returned to the system. Its connections (which will be discussed later) will be closed. Destruction of a task also implies destruction of all the descendants of the task, i.e., those tasks it created, those they created, etc. The _Destroy primitive is synchronous, which means that all this will be completed before the primitive returns, so the resources are immediately available for reuse.

COMMUNICATIONS AND SYNCHRONIZATION

Although the main purpose of using tasks is to allow them to proceed with their own computations independently and in parallel, each running at its own speed, in most practical programs the tasks need to communicate and synchronize with each other occasionally. In particular, if more than one task accesses the same location in storage, and if the value at that location can change, then there is a possibility of a critical race where the result of the program depends on when which task reads and when which task writes the value. Various systems have provided different mechanisms for dealing with this problem, including semaphores, monitors, conditional critical regions, rendezvous, and several kinds of message passing. Harmony addresses the problem by a paradigm for structuring multitask programs, using four primitives that provide message passing. A message, in Harmony, is a variable length contiguous block of storage, the first two bytes of which are a short unsigned integer specifying the length. Typically the type of the message is actually a struct or union, but the primitives declare the type for their arguments as pointer to char, requiring a cast. The primitives are implemented as functions:

```
id = _Send( rqst, rply, id );
id = _Receive( rqst, id );
id = _Try_receive( rqst, id );
id = _Reply( rply, id );
```

The semantics of these primitives are as follows: A task wanting to send a message to another task sets up that message in space pointed to by

the rqst argument, sets up space pointed to by the rply argument into which to receive a message in reply, and then calls the _Send primitive specifying in the id argument the id of the desired correspondent task. A task wanting to receive a message must set up space pointed to by the rqst argument into which the contents of the sender's rqst message can be copied, and then calls either the _Receive or the _Try_receive primitive, specifying in the id argument either the id of a particular correspondent task or 0, the latter indicating that message will be accepted from any task (in fact FIFO). When the _Receive or _Try_receive primitive returns, the value id returned by the function is the id of the correspondent task, and the sender's message will have been copied into the receiver's message. After the receiving task has finished processing the sender's request, it sets up in space pointed to by the rply argument the message to be replied back to the sender, and calls the _Reply primitive with the id argument specifying the sender. Specifying the sender is necessary because it is not required that a sender be replied to before receiving another message, nor that messages be replied to in the order they were received, in the inverse order to which they were received, or any other fixed order. A reply can only be made to a task from which a message has been received and to which a reply has not yet been made, however. The message being replied to the sender is copied from the receiver's space to the space provided by the sender before the _Send primitive returns.

A particularly important aspect of the semantics of these primitives is their blocking behaviour, for this is how tasks synchronize with each other. The sender blocks from when the _Send primitive is called, not just until the corresponding _Receive or _Try_receive primitive is called by the receiver but until the receiver calls the corresponding _Reply primitive. A receiver calling the _Receive primitive blocks if the id argument is 0 and no task has already called the _Send primitive to send a message to this receiver, or if the id specifies a particular task and that task has not yet called the _Send primitive to send a message to this receiver. The receiver calling the _Receive primitive unblocks when the message for which it is waiting becomes available. By contrast, the _Try_receive primitive is nonblocking and fails immediately, returning 0, if the desired message is not available when the receiver calls the primitive. The _Reply primitive is a nonblocking operation for the receiver (because the sender is known to be already blocked) and it also unblocks the sender, allowing it to run again. Of course if the sender and receiver happen to run on the same processor, the normal scheduling rules will dictate which actually runs first. Nothing else in the foregoing discussion is affected by whether the communicating tasks happen to be on the same processor or different processors.

Because tasks can be destroyed, it is possible that the primitives can fail. This is the reason _Send and _Reply are also implemented as functions: all four primitives return the id of the correspondent task if

they succeed, and 0 if they fail. Another point to note is that the length of the message indicated in the space copied from and copied to may disagree. Only the shorter length is actually copied, but since the length copied from is the first thing copied into the space copied to (overwriting the length originally there), it is possible to find out if the message was truncated, and how much space would be required to get it all.

There is a certain symmetry between _Send and _Reply in that both transfer data to another task. By turning around the usual role of these primitives, a nonblocking communications is possib'e between one task and another which has prearmed for the communication by sending to indicate it is ready to receive a message. That is, suppose task A has some data to transfer to task B. Normally, one would think of A sending a message to B. If this is done, A will block until B replies. If, however, it is important that A not block, the communication could be reversed and B could instead send to A and block. When A is ready with the data, it can transfer those data to B in a reply, with which A will not block but will continue to execute, and B will also resume execution in parallel. When B finishes processing the data, it can send to A again, returning any results and preparing to repeat the cycle.

The paradigm is that, in general, tasks do not use resources directly; for example, several tasks will not all access a location in memory that can change. Instead, each such resource is "owned" by a single task, and operations on the resource are performed by that task on behalf of other tasks that have sent messages requesting these operations. The task given the responsibility for some resource is referred to as a "server", since it is seen by other tasks as providing a service - the tasks using this service are referred to as "clients". Mutual exclusion follows directly from the fact that the server is a single task, with a single thread of execution, and messages are received one at a time. Sometimes, parallelism can be exploited with respect to the resource the server controls. This is done by the server having worker tasks that employ the reverse communication described earlier - the worker tasks send to the server whenever they have completed the last item of work they have been given, and are available to be given (by a reply) another item of work to do.

Occasionally, it is appropriate for closely cooperating tasks to share memory, because the protocol by which they cooperate ensures this is safe, and because the overheads of passing the data back and forth by message passing would be excessive. Harmony supports this, in that addresses mean the same to all tasks, whether on the same processor or not, so externals can be shared by tasks on the same processor, and pointers can be passed in messages and used by recipient tasks. This practice requires great care and understanding, however, and should be avoided where possible.

The semantics of the call to _Send used by a client task to a server are identical to a procedure call, and so are familiar and comfortable to all programmers. Because _Send and _Receive can block, there is some possibility of deadlock, but in practice it does not happen when this paradigm is used because almost the only way to get a deadlock would be to have a cycle of tasks sending to each other. With a hierarchy of services implemented using lower level services, this never happens.

STORAGE CLASSES

A program in C running under Harmony has access to five storage classes: external, static, auto, register, and dynamically allocated.

The first two can be useful, but have complicated aspects. Because the executable image for each processor is linked separately, externals or statics referenced by tasks on one processor are different from those referenced by tasks on another, even though the name is the same. On the other hand, all tasks on the same processor will access the same instance of a named external or static - which, unless the external or static is initialized and not subsequently assigned to, can lead to critical races as noted previously. It is also worth noting that ROMability is often an objective of realtime systems, and that an external or static that is initialized by compile-time initialization is not ROMable if it is subsequently assigned to.

The situation is much simpler for variables of the auto storage class. They are allocated on the stack of the task, dynamically as the task executes. There is a different instance at each function call, never mind for each task. The stack for the task itself is dynamically allocated when the task is created and recovered when the task is destroyed. Because local memory typically has faster access, the stack for a task is allocated from memory local to the processor on which the task will run.

Some of the hardest bugs to find involve stack overflow, where the stack required at some point in the nesting list is bigger than what was allocated for the task. Unfortunately, there is rarely adequate hardware checking for this, and many compilers, including Whitesmiths', do not generate code for a runtime software check on function entry. Harmony provides two aids to alleviate this unsatisfactory situation. The first is a function that can be called to determine if the stack for this task is already overflowed,

```
boolean = _Stackoverflow();
```

The other is a program development tool, bound, that determines the

required stack size when the call graph for a task is simple (an acyclic digraph of known functions) and interactively aids the user in choosing an appropriate stack size when there is recursion or indirect function calls.

The register storage class of course has no problems for tasks.

Storage can be dynamically allocated in that a contiguous block of storage is obtained by a call to the primitive

```
ptr = _Getvec( size );
```

where size is the number of bytes in the desired block, and the value returned is typed as pointer to char, so typically must be cast to the required type. The value 0 is returned if the allocation failed. Memory is allocated from a pool local to the processor executing the primitive. The memory obtained is not initialized to anything particular.

When a block of storage is no longer required, it can be returned to the storage pool by calling the primitive

```
_Freevec( ptr );
```

When a task is destroyed, all of its allocated blocks are automatically returned to the storage pool.

Occasionally it is desirable to allocate an overly generous block of storage, then return the excess to the storage pool once the true requirements are known. This is achieved by calling the primitive

```
_Trimvec( ptr, size );
```

STREAM INPUT/OUTPUT

The model of stream I/O is a powerful and uniform abstraction that is useful for doing input/output to a wide range of devices, from serial lines to terminals to tapes to disk files. A stream is an infinite sequence of bytes in which a current position is defined, and on which the operations of getting the next byte, putting the next byte, and changing the current position may be performed. Any higher level structure must be inferred from the value of the current byte, the context of the current byte, or the current byte position. For example, numbers are formed as sequences of bytes separated by delimiters; variable length lines of text are readily supported by separating the lines by a new-line symbol; end of file is indicated by an end-of-file symbol; and transparent communications line control is achieved by ASCII control characters and escapes inserted into the text to be transmitted.

Harmony supports connection oriented stream I/O. (All types and manifest constants referenced below are declared in the standard header, sys.h, which should be included in any compilation of a Harmony program.) A stream is made accessible by being opened by a task,

```
ucb = _Open( pathname, mode );
```

where pathname is a string that symbolically defines the stream of interest and will be discussed more fully later, and mode is a short int with specific values READ, WRITE, or READWRITE indicating how the stream is to be used. The value returned, ucb, points to a dynamically allocated struct called a user connection block that defines the buffer, server, and connection number associated with this stream - details of which the user need not be aware. The value 0 is returned if the open failed. There is no fixed limit as to how many streams may be open. In general, the same stream may be opened more than once (although some servers may disallow this), each with its own current position. This is typically done when several tasks use the same stream.

A stream which no longer needs to be accessible should be closed:

```
_Close( ucb );
```

Destruction of a task automatically closes all its open streams. It should be noted that closing a stream does not imply flushing any buffers (see below) and this must be done explicitly.

For each task, one of the opened streams is selected for input at any time, and all input comes from this stream until the selection is changed by

```
prev_ucb = _Selectinput( ucb );
```

Similarly, for each task one of the opened streams is selected for output, and all output goes to that stream until the selection is changed by

```
prev_ucb = _Selectoutput( ucb );
```

In this way, input bytes can be obtained from the selected input stream without the cost and inconvenience of specifying the stream each time

```
byte = _Get();
```

and similarly output bytes can be put to the selected output stream without the cost and inconvenience of specifying the stream each time

```
byte = _Put( byte );
```

The byte output is returned as the value of _Put because of the convenience this often provides in program control structure.

Although the abstraction of stream I/O is attractive, it would be impractically expensive if it were not actually buffered, so that _Get and _Put usually just reference a local buffer instead of actually doing I/O or even sending requests to a server. This buffering is transparent on input (although it makes finding whether a byte is available for input impossible without blocking) but not on output. The output buffer will be sent to the server to be output when it is full, but situations often arise where the buffer should be forced out without waiting to be full. This is done by

```
_Flush();
```

(It is unfortunate that the phrase "flush the buffer" has two conflicting interpretations - instead of the one used here, there is a different one sometimes used implying that the buffer contents should be discarded.)

The buffered input makes possible a function which is very convenient when breaking input tokens at delimiters, or when reading input that may be of several different forms. This is the action of putting the last byte read back into the input stream, so it will be read again by the next _Get. This action will work for any input stream, but it only works for the last byte read.

```
_Unget();
```

For a more general mechanism, it is meaningful for some streams to change the current position to an arbitrary position in the stream, either indexed absolutely from the origin of the stream, or relative to the current position.

```
_Seek( ucb, position, relative );
```

Here position is a long int, and relative is nonzero if the indexing is relative to the current position. (THIS FUNCTION IS NOT YET IMPLEMENTED.)

On top of this basis, a few higher level functions are provided for convenience. Numbers are read free format, with the C convention where

0xNNN represents hex, 0NNN octal, and NNN with nonzero leading digit decimal, by

```
number = _Getnum();
```

Numbers can be printed in decimal by

```
_Putdec( number );
```

or in hexadecimal by

```
_Puthex( number );
```

and strings can be printed by

```
_Putstr( string );
```

This can all be combined by

```
_Printf( format, item1, item2, ... );
```

which prints the variable number of items according to the format string. As in Unix, the format characters are directly output, except that %% is printed as %, %c prints the corresponding item as a character, %i prints the corresponding item as a decimal integer, %h prints the corresponding item as a hexadecimal integer, and %s treats the corresponding item as a pointer to a string and prints that. Full format control as in Unix is not supported. In particular, there is currently no floating point conversion.

OTHER INPUT/OUTPUT, INTERRUPTS, AND SERVERS

Not all I/O devices fit the stream I/O abstraction, and even for those which can be forced to fit, there can be realtime applications that cannot afford to pay the performance penalty of a uniform implementation. Consequently it is necessary for Harmony to allow the user to design and implement his own I/O abstractions, and for tasks in a user program to have the direct ability to perform I/O operations and to handle interrupts. Even when the stream I/O abstraction is used, these facilities may be needed because since Harmony is an open system, the user may have to supply the tasks supporting the streams to and from his particular devices. At the very least the configuration of a control computer for realtime applications varies so widely that the user will have to specify which of the available servers are needed in any particular program.

For this reason, Harmony ignores the supervisor versus user state distinction present in most hardware, and runs the whole program in supervisor state. For memory mapped I/O, such as the Motorola 68000 uses, a task owning an I/O device can do the I/O instructions directly in C - other processors which use special I/O instructions would require assembler subroutines. The actual I/O commands depend, of course, on the device, and might involve either programmed I/O or DMA. Synchronizing the task with the I/O device can be done busywait by reading status registers, either pre-busywait checking that the device is ready before issuing an I/O command, or post-busywait waiting for the device to complete after an I/O command is issued before going on. There are situations where busywait synchronization is the preferred solution, because the busy state is guaranteed to clear in less time than would be required for a context switch to another task. However, in general it is unsatisfactory because in uselessly consuming processor cycles it prevents other tasks of the same or lower priority from running, because it is difficult to do with more than one device (especially input devices), and because it can readily lead to deadlocks. Generally, better practice is for the task to initiate an I/O command to the device, to synchronize by blocking while waiting for an interrupt to signal completion of the command, then to retrieve any results. (A pre-wait version, similar to pre-busywait, is occasionally possible instead.) This action is something like the blocking that occurs with the _Send primitive, but the primitive Harmony provides for it,

```
_Await_interrupt( interrupt_id, rply_msg );
```

is at a somewhat lower level, assuming the controlling task will issue the I/O command and retrieve the results. (The design of some controller hardware can actually force different tasks to be used to issue the I/O command and to await the interrupt.) The rply_msg argument of the primitive is provided because experience has shown that many peripheral devices have data which are volatile and must be captured at the instant the interrupt is first detected, for they will be lost by the time the waiting task is activated. It is important to note that the way this primitive is defined, at most one task can be waiting for any specific interrupt, a task can only be waiting for one interrupt, the waiting task must run at the Harmony priority corresponding to the hardware level at which the interrupt will occur, and, of course, the waiting task must run on the processor physically connected to the interrupt.

It would be nice to go down to only this level of abstraction, but since Harmony is an open system, indeed because it is a configurable system, it is often necessary to descend further and implement the second level interrupt handler. (The first level interrupt handler is invoked by the actual hardware interrupt, saves the registers on the stack of the

interrupted task, and transfers to the proper second level interrupt handler for the particular processor interrupted.) The second level interrupt handler must save the stack pointer in the task descriptor of the interrupted task, must demultiplex the hardware interrupt by finding which interrupt source connected to that hardware interrupt actually caused the interrupt, must clear the interrupt while recording any volatile data, must find whether there was a task waiting for the interrupt and if so activate it (passing it the volatile data), or if not reactivate the interrupted task (first incrementing the external _Spurcount and recording the interrupt id in the external _Spurdev). The second level interrupt handler is code written in assembler, and no stack is available as the stack of the interrupted task cannot be used and the task to dispatch has not been determined yet. Interrupts are disabled throughout this code. Note that the concept of spurious interrupts here includes not just those caused by noise or by configuration errors, but also those where no task is yet awaiting the interrupt - a protocol error. The interrupt id is actually the byte offset in the vector _Int_table at which the id of the waiting task, if any, is stored.

The machinery just described is sufficient to provide direct control over any physical device. It should be recognized, however, that the task structure of Harmony facilitates creating "smart peripherals" where software implemented as a controlling task provides an interface closer to the level of device actually desired, instead of what the raw hardware provides. Not only do the messages serviced by this controlling task define a higher level device to client tasks, but if the controlling task runs on a different processor from that of the client, the latter sees the computational load offloaded too. Eventual migration of the "smart peripheral" task into specialized hardware is also made more convenient if that hardware uses the same interface. An example using a conventional computer peripheral would be a "file store device" implemented by a task controlling an ordinary disk, where the task also did space allocation, catalog management, etc. so that the operations on the smart device were to create and destroy linear address spaces, grow or shrink them as necessary, and read or write them - independent of physical block location, let alone cylinder boundaries, bad tracks, etc. An even higher level would be a "file system device" implemented by a task using the "file store device", where this task provided a heirarchical file system with access control, concurrency control, etc. Another example, this time from robotics, would be a "vision system device" implemented by a task controlling a TV camera and framegrabber, where the task responded to client tasks by indicating objects seen, together with their locations and orientations.

Normally, a client task wanting to do I/O does it by exchanging messages with the server task which supports that I/O model for the desired device. The types of messages and the protocol of how they are used are

what really define an I/O model; we will look at those for the stream I/O model shortly. Many server tasks may support a given model. The server task may be the task directly controlling the hardware device or, as discussed above, a task implementing some higher level abstraction. The client task uses a symbolic name to establish a connection with the appropriate server. This provides "device independence" within the class of servers supporting that I/O model. It does this by calling _Open, which sends a message to the directory task. The directory task matches the first component of the name (at most 30 characters, terminated by :) against the list of available servers and, if a match is found, sends the open message on to that server. (The directory task initially has no built-in knowledge of which servers are available, so any server must send a REPORT_FOR SERVICE message to the directory task reporting the symbolic name by which it is known.) The server sets up the connection or can deny it, based on the rest of the name, on what connections already exist, etc. (The rest of the name might be interpreted as a directory and file pathname by a file system server, or a telephone number by an automatic calling unit server, and so forth.) The server replies to the directory with a number identifying this connection, and the id of the task the client should communicate with for this connection (normally the server itself, but possibly an agent task instead), as well as other parameters such as buffer size and flags. The directory then replies this to the client. When the connection is used subsequently, messages from the client must identify the connection, for a server may simultaneously be supporting multiple connections, even to the same client task - the file system task might open several files for the same client task, for instance. A connection is closed by the client task sending a close message to the server; this will be done automatically for any connections still open when a client task is destroyed.

An I/O server task thus must report for service to the directory task and must be prepared to receive open and close messages. What other messages it handles depend on the I/O model supported and on the device controlled. Stream I/O, for instance, implies handling a read message to provide another buffer of input, a write message to output another buffer, a flush message to ensure that any buffers associated with the connection are flushed, and a seek message to move the current position in the stream. A server supporting this I/O model for a serial line, however, would probably ignore the seek message. Several examples of servers are included in Harmony, including servers to support interactive terminals and to act as virtual terminals to remote computers.

USER'S VIEW OF A PROGRAM

Recapitulating and extending what was said above, a user sees a program as a set of processor images, one for each processor in the

hardware configuration. A processor image is obtained by linking three fundamental user supplied externals with the Harmony kernel and support libraries, as well as with whatever user supplied externals and functions are brought in, directly or indirectly, by the references in the three fundamental externals.

The three fundamental externals for a processor image are

```
unsigned _Pnumber =
```

which must be initialized with a value specifying which processor this image is for;

```
void (*(_Intvec[7]))() =
```

which must be initialized as a vector of seven entry points for the second level interrupt handlers required for this processor (on a 68000, only the lowest five levels may be user supplied - this would be slightly different for other implementations); and finally

```
struct TASK_TEMPLATE _Template_list[] =
```

which must be initialized as a vector of all task templates for tasks that can be created on this processor (the vector must be terminated by a null task template, i.e., one in which all fields are 0).

On processor 0, three special task templates must be included in _Template_list. A task template with GLOBAL_INDEX of 1 must be provided as the first user task, which the system starts and for which the root function is normally the user function main(). This is the start of the user's program. A task template with GLOBAL_INDEX of 2 must also be provided as the directory task, for which the root function would normally be _Directory, but which might conceivably have other than the default priority of 5. Finally, a task template with GLOBAL_INDEX of 3 must also be provided as the gossip task, for which the root function would normally be _Gossip, but which might conceivably have other than the default priority of 5.

The user is responsible for creating whatever servers his program will require. This is best done at the start of function main(), in the first user task. Care must be taken to allow for the critical race where the server must report for service to the directory task before the first client task can open a connection with it.

In choosing how to structure the source code, it is wise to remember

that all compilation source should have an include file sys.h, in order to have necessary struct and manifest constant declarations. It is also good practise to include a file giving all template index assignments to be used throughout the whole program. Remember to include the templates for all servers that will be used, as well as for the worker tasks they create. When choosing identifiers, note that Harmony uses the convention that all Harmony-defined objects, both internal and visible, start with an underscore in order that accidental name conflicts can be avoided and system use can be recognized. Another convention is that struct tags, field names, and manifest constants are all defined by upper case only names, and that externals and functions have names for which the first alphabetic is upper case and there is at least one lower case alphabetic.

Linker parameters are different for processor images for different processors, which is most conveniently handled by working with libraries, making linking an explicit step in program building, and using a specific command file for linking processor images for each processor. On the Chorus machine each processor is defined with RAM starting on a megabyte boundary. The processor image is linked with that as base. The first routine loaded is forced to be _Setup, or _Setup0 on processor 0, and control transfers there to start execution. This routine sets up the C environment, then calls the Harmony kernel to build and initialize Harmony. Processor 0 initializes its Harmony kernel first, then opens the multiprocessor gates so the other processors can initialize their Harmony kernels. (A gate is a location used for synchronization.) In a downloading situation, on reset all processors other than processor 0 close their multiprocessor gate, then enter a busywait loop waiting for their multiprocessor gate to open, and processor 0 executes whatever code is necessary to accept a download of code first for the other processors, and then for itself. The end of downloading causes it to execute _Setup0. The linker must define the symbol end which is used as beginning of the storage pool for that processor.

SUPPORT TOOLS

Harmony currently provides four support tools: listing, examine, bound, and debug. These are all at a very primitive stage of development. The first three, i.e., listing, examine, and bound, all run in the development environment rather than in Harmony itself; examine and bound are intimately involved with the exact form of a stored executable image and with the generated code from the current compiler. Consequently these tools may need considerable conversion to be used outside the environment in which Harmony was initially developed. Nevertheless, they do serve as prototypes for tools that are useful.

listing

The style of programming encouraged by Harmony consists of many small functions, each stored in a separate file, and many header files, each defining structs and macros associated with a different abstraction. Compilations are done by compiling include files that reference the appropriate source. A readable listing needs to be arranged as a book, with cover, table of contents, numbered pages with headings, and a concordance or index at the end. This tool, given an include file, produces this.

examine

It is sometimes necessary to examine a bound executable image, to determine overall characteristics such as the size of the text or data segments, to look up external symbols, to disassemble instructions or display initialized data, or even to patch erroneous values. This tool, given a .exe file produced by Whitesmiths' linker, does this.

bound

Bounding the stack required for a given task involves discovering the call graph of procedures called from the root function, or called by procedures called from the root function, etc. The stackframe size required for each of these procedures must be determined. All possible nesting lists can then be simulated to determine the maximum stack size required. Two things force this to be an interactive, rather than a wholly automatic, process. First, sometimes there are indirect procedure calls, i.e., calls to procedures which are formal arguments of the calling procedure or which are values returned by calls to other procedures or which are values of procedure-type variables. These cannot be known without running the program, so the possible procedures must be supplied interactively. Second, the call graph may involve a cycle, implying possible recursion, and while the stack requirement per cycle can be made available, the number of levels of recursion to support must be supplied interactively. This tool, given an .exe file, allows the user to determine stack sizes for several root functions in that file.

debug

The question of how to debug a multitask program is still very much unanswered and may even require hardware support. For now, this very simple debugger is all that is available. Breakpoints cannot even be planted, but must be compiled into the source as calls to the function _Breakpoint. Values to be inspected must be identified by absolute

address, and values are only available in decimal or hexadecimal. On the other hand, the debugger can print certain known system data structures, such as task descriptors, and works with interrupts enabled and with breakpoints on code in processors other than the one with the terminal. Considerable evolution can be expected. The current version requires server "DEFAULT:" to be available.

PERFORMANCE

It is difficult for a portable system to make quantitative statements about performance, since measured values depend upon tuning of the kernel code, which compiler is chosen, the microprocessor instruction set, the clock speed, the memory used, other aspects of the processor board design, etc. Nevertheless, qualitative statements about performance are important, to make sensible design tradeoffs in application programs. The measurements described here were made using a 10 MHz 68000 with no wait states, Harmony having been compiled by Whitesmiths' C compiler.

The first important characteristic is response time to an interrupt, i.e., delay from the moment of interrupt until the _Await_interrupt of the waiting task returns. This is measured as 0.100 ms. An indication of how good this really is can be seen from the fact that the fastest assembly language interrupt handler, operating outside the tasking abstraction but saving the state of the interrupted task and sorting out which processor's interrupt vector to use, takes 0.040 ms.

The second important characteristic is the send-receive-reply cycle time. When both tasks are running in the same, otherwise empty, processor, this is measured as 1.15 ms. When the two tasks are running in different, otherwise empty, processors, this is measured as 1.10 ms, the reduction coming about because of parallelism in execution of the primitives on the two processors. Also important are how much processor time is required on the sending processor and how much on the receiving processor. These are measured as 0.460 ms and 0.720 ms, respectively.

An important aspect of the message passing implementation is that it is distributed. Only the processor or processors that the communication tasks are running on participate in the protocol: there is no central "switching" processor involved. The actual bus cycles used on the shared bus are so few that contention for these will not be a bottleneck. Message passing between tasks on the same processor is treated uniformly with message passing between tasks on different processors and not by special code.

The measured times given above are quite commensurate with the best

task context switching times achieved by other systems on similar hardware, even uniprocessor systems.

In some situations, however, they are not fast enough. This problem has been recognized as being due to inadequate hardware support for operating system primitives, and one solution often proposed is to provide an operating system coprocessor for each processor or to provide microcode support for the primitives. Although this has not yet been tried for Harmony, it has been very effective for similar systems, providing more than an order-of-magnitude speedup. Consequently, the message passing protocol for Harmony is implemented by a single identifiable piece of code that should be moved to a coprocessor or to microcode if that is available.

REFERENCES

1. W.M. Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. Software Pract. Exper. 11; 435-466; 1981.

2. W.M. Gentleman. If only the hardware... (A software designer's lament). Proceedings of the IEEE International Workshop on Computer Systems Organization, New Orleans, LA. March 29-31, 1983. pp. 88-95.

APPENDIX A. A SAMPLE MULTITASK PROGRAM

The following program is a simple multitask program which illustrates many of the ideas and primitives discussed in this manual. In this Appendix, the program is run on only processor 0. The function main is the root function for the first user task. It creates a server for a terminal attached to a serial port and opens this for input and output. It then creates a child task and goes into an infinite loop, sending to the child task and printing the response and accompanying text to the selected terminal.

```
#include        "[harmony.src]sys.h"
#include        "[harmony.src]templates.h"

#define CHILD           20
 unsigned _Pnumber = 0;

struct TASK_TEMPLATE _Template_list[] =
  {
      { 1, main, 180, 7, 0 },
      { 2, _Directory, 268, 7, 0 },
      { 3, _Gossip, 226, 7, 0 },
      { CHILD, Child, 108, 6, 0 },
      { POACIA0, _POAcia0_server, 412, 5, 0 },
      { TTI, _Tti, 100, 0, 0 },
      { TTO, _Tto, 126, 0, 0 },
      { 0, 0, 0, 0, 0 }
  };

void (*(_Intvec[7]))() =
  {
    0,              /* int lvl 1 for this processor, Harmony priority 4 */
    0,              /* int lvl 2 for this processor, Harmony priority 3 */
    0,              /* int lvl 3 for this processor, Harmony priority 2 */
    _Ptm_int,       /* int lvl 4 for this processor, Harmony priority 1 */
    _Serial_int,    /* int lvl 5 for this processor, Harmony priority 0 */
    0, 0            /* remaining levels must be zero, will be plugged */
  };

main()
  {
    unsigned    child;
    struct STD_RQST     request;
    struct STD_RPLY     reply;

    _Create( POACIA0 );
    _Selectinput( _Open( "POACIA0:", READ ) );
    _Selectoutput( _Open( "POACIA0:", WRITE ) );

    _Putstr( "parent creates child\n" );
    child = _Create( CHILD );

    for(;;)
      {
        request.MSG_SIZE = sizeof( request );
        reply.MSG_SIZE = sizeof( reply );
        _Send( (char *)&request, (char *)&reply, child );
```

```
        _Putstr( "child replied " );
        _Putdec( reply.RESULT );
        _Putstr( "\n" );
      }
  };

Child()
  {
    unsigned counter, requestor;
    struct STD_RQST request;
    struct STD_RPLY reply;

    counter = 0;

    for(;;)
      {
        request.MSG_SIZE = sizeof( request );
        requestor = _Receive( (char *)&request, 0 );

        reply.MSG_SIZE = sizeof( reply );
        reply.RESULT = counter++;
        _Reply( (char *)&reply, requestor );
      }
  };
```

APPENDIX B. A SAMPLE MULTITASK MULTIPROCESSOR PROGRAM

This program is the same as the program of Appendix A, but the child task has been moved to run on processor 1. Note that the only difference in the program is that the task template for the child task has been moved from the _Template_list for processor 0 to the _Template_list for processor 1. The text for the root function Child has been moved to the second file also, since it is being compiled with the three data structures that define the executable image. If it were being referenced from a library, as the other functions are, this would be unnecessary.

FILE ZERO:

```
#include        "[harmony.src]sys.h"
#include        "[harmony.src]templates.h"

#define CHILD           20

unsigned _Pnumber = 0;

struct TASK_TEMPLATE _Template_list[] =
  {
      { 1, main, 180, 7, 0 },
      { 2, _Directory, 268, 7, 0 },
      { 3, _Gossip, 226, 7, 0 },
      { POACIA0, _POAcia0_server, 412, 5, 0 },
      { TTI, _Tti, 100, 0, 0 },
      { TTO, _Tto, 126, 0, 0 },
      { 0, 0, 0, 0, 0 }
  };

void (*(_Intvec[7]))() =
  {
    0,              /* int lvl 1 for this processor, Harmony priority 4 */
    0,              /* int lvl 2 for this processor, Harmony priority 3 */
    0,              /* int lvl 3 for this processor, Harmony priority 2 */
    _Ptm_int,       /* int lvl 4 for this processor, Harmony priority 1 */
    _Serial_int,    /* int lvl 5 for this processor, Harmony priority 0 */
    0, 0            /* remaining levels must be zero, will be plugged */
  };

main()
  {
    unsigned    child;
    struct STD_RQST     request;
    struct STD_RPLY     reply;

    _Create( POACIA0 );
    _Selectinput( _Open( "POACIA0:", READ ) );
    _Selectoutput( _Open( "POACIA0:", WRITE ) );

    _Putstr( "parent creates child\n" );
    child = _Create( CHILD );

    for(;;)
      {
        request.MSG_SIZE = sizeof( request );
```

```
        reply.MSG_SIZE = sizeof( reply );
        _Send( (char *)&request, (char *)&reply, child );

        _Putstr( "child replied " );
        _Putdec( reply.RESULT );
         Putstr( "\n" );
    }
};
```

FILE ONE:

```
#include        "[harmony.src]sys.h"

#define CHILD           20

unsigned _Pnumber = 1;

struct TASK_TEMPLATE _Template_list[] =
  {
      { CHILD, Child, 108, 6, 0 },
      { 0, 0, 0, 0, 0 }
  };

void (*(_Intvec[7]))() =
  {
    _VG121_int,    /* int lvl 1 for this processor, Harmony priority 4 */
    0,             /* int lvl 2 for this processor, Harmony priority 3 */
    0,             /* int lvl 3 for this processor, Harmony priority 2 */
    _Ptm_int,      /* int lvl 4 for this processor, Harmony priority 1 */
    _Serial_int,   /* int lvl 5 for this processor, Harmony priority 0 */
    0, 0           /* remaining levels must be zero, will be plugged */
  };

Child()
  {
    unsigned counter, requestor;
    struct STD_RQST request;
    struct STD_RPLY reply;

    counter = 0;

    for(;;)
      {
        request.MSG_SIZE = sizeof( request );

        requestor = _Receive( (char *)&request, 0 );

        reply.MSG_SIZE = sizeof( reply );
        reply.RESULT = counter++;
        _Reply( (char *)&reply, requestor );
      }
  };
```

APPENDIX C. DECLARATIONS

```
char *_Getvec( size )
    unsigned size;

_Freevec( block )
    char *block;

_Trimvec( block, size )
    char *block;
    unsigned size;

unsigned _Create( task_index )
    unsigned    task_index;

_Destroy( id )
    unsigned    id;

void_Suicide()

unsigned _My_id()

unsigned _Father_id()

unsigned _Send( rqst_msg, rply_msg, id )
    char        rqst_msg[], rply_msg[];
    unsigned id

unsigned _Receive( rqst_msg, id )
    char rqst_msg[];
    unsigned id;

unsigned _Try_receive( rqst_msg, id )
    char rqst_msg[];
    unsigned id;

unsigned _Reply( rply_msg, id )
    char rply_msg[];
    unsigned id;

_Await_interrupt( interrupt_id, rply_msg )
    unsigned interrupt_id;
    char rply_msg[];

int _Stackoverflow()
```

```
struct UCB *_Open( name, mode )
    char *name;
    short int mode;

_Close( ucb )
    struct UCB *ucb;

struct UCB *_Selectinput( ucb )
    struct UCB *ucb;

struct UCB *_Selectoutput( ucb )
    struct UCB *ucb;

_Get()

_Put( byte )
    char byte;

_Flush()

_Unget()

_Getnum()

_Putdec( n )
    int n;

_Puthex( n )
    int n;

_Putstr( s )
    char *s;

_Printf( fmt, x )
    char *fmt;
    int  x;
```

APPENDIX D. SUPPLIED TASKS

_Directory	root function for the directory task used to establish connections between clients and servers
_Gossip	root function for the general logging and reporting task, used in particular to report processor faults
_P0Acia0_server	root function for a terminal server using acia 0 on pnumber 0
_P0Acia1_server	root function for a terminal server using acia 1 on pnumber 0
_Ehvt_server	root function for an echo handshake virtual terminal server using acia 1 on pnumber 1
_Tti	root function for task receiving bytes from an acia
_Tto	root function for task transmitting bytes to an acia